

Parse Tree

PROGRAM

```
<p4program> := <declarationList>

<declarationList> := { <declaration> | ; /* empty declaration */ }*

<declaration> := <variableDeclaration> | <externDeclaration> | <actionDeclaration> |
    <typeDeclaration> | <parserDeclaration> | <controlDeclaration> | <instantiation> |
    <errorDeclaration> | <matchKindDeclaration> | <functionDeclaration>

<nonTypeName> := IDENTIFIER | apply | key | actions | state | entries | type

<name> := <nonTypeName> | TYPE_IDENTIFIER

<nonTableKwName> := IDENTIFIER | TYPE_IDENTIFIER | apply | state | type

<parameterList> := { <parameter> { , <parameter> }* }

<parameter> := { <direction> } <typeRef> <name> { = <expression> }

<direction> := in | out | inout

<packageTypeDeclaration> := package <name> <optTypeParameters> ( <parameterList> )

<instantiation> := <typeRef> ( <argumentList> ) <name> ;

<optConstructorParameters> := { ( <parameterList> ) }
```

PARSER

```
<parserDeclaration> := <parserTypeDeclaration> <optConstructorParameters>
    { <parserLocalElements> <parserStates> }

<parserLocalElements> := { <parserLocalElement> }*

<parserLocalElement> := <variableDeclaration> | <instantiation> /* | <valueSetDeclaration> */

<parserTypeDeclaration> := parser <name> <optTypeParameters> ( <parameterList> )

<parserStates> := <parserState> { <parserState> }*

<parserState> := state <name> { <parserStatements> <transitionStatement> }

<parserStatements> := { <parserStatement> }*

<parserStatement> := <assignmentOrMethodCallStatement> | <directApplication> | <parserBlockStatement> |
    <variableDeclaration> | <emptyStatement>

<parserBlockStatement> := { <parserStatements> }

<transitionStatement> := { transition <stateExpression> }

<stateExpression> := <name> ; | <selectExpression>

<selectExpression> := select ( <expressionList> ) { <selectCaseList> }

<selectCaseList> := { <selectCase> }*

<selectCase> := <keysetExpression> : <name> ;
```

```

<keysetExpression> := <tupleKeysetExpression> | <simpleKeysetExpression>

<tupleKeysetExpression> := ( <simpleExpressionList> )

<simpleExpressionList> := <simpleKeysetExpression> { , <simpleKeysetExpression> }*

<simpleKeysetExpression> := <expression> /* { ( mask | range ) <expression> } */ | default | _

/* <valueSetDeclaration> := valueset '<' ( <baseType> | <tupleType> | <typeName> ) '>'
   '(' <expression> ')' <name> ';' */

```

CONTROL

```

<controlDeclaration> := <controlTypeDeclaration> <optConstructorParameters>
    { <controlLocalDeclarations> apply <blockStatement> }

<controlTypeDeclaration> := control <name> <optTypeParameters> ( <parameterList> )

<controlLocalDeclaration> := <variableDeclaration> | <actionDeclaration> | <tableDeclaration> |
    <instantiation>

<controlLocalDeclarations> := { <controlLocalDeclaration> }*

```

EXTERN

```

<externDeclaration> := extern
    ( <nonTypeName> <optTypeParameters> { <methodPrototypes> } ) |
    ( <functionPrototype> ; )

<methodPrototypes> := { <methodPrototype> }*

<functionPrototype> := <typeOrVoid> <name> <optTypeParameters> ( <parameterList> )

<methodPrototype> :=
    ( <functionPrototype> ; ) |
    ( TYPE ( <parameterList> ) ; ) /* constructor */

```

TYPES

```

<typeRef> := <baseType> | <namedType> | <tupleType>

<namedType> := <typeName> | <specializedType> | <headerStackType>

<prefixedType> := { . } TYPE_IDENTIFIER

<typeName> := <prefixedType>

<tupleType> := tuple < <typeArgumentList> >

<headerStackType> := <typeName> [ <expression> ]

<specializedType> := <typeName> < <typeArgumentList> >

<baseType> :=
    bool | error | string | void |
    int { < <integerTypeSize> > } |
    bit { < <integerTypeSize> > } |
    varbit < <integerTypeSize> >

<integerTypeSize> := INTEGER /* | '(' <expression> ')' */

<typeOrVoid> := <typeRef> | void | IDENTIFIER /* type variable */

```

```

<optTypeParameters> := { < <typeParameterList> > }

<typeParameterList> := <name> { , <name> }*

<realTypeArg> := _ | <typeRef>

<typeArg> := _ | <typeRef> | <nonTypeName>

<realTypeArgumentList> := <realTypeArg> { , <realTypeArg> }*

<typeArgumentList> := { <typeArg> { , <typeArg> }* }

<typeDeclaration> := <derivedTypeDeclaration> | <typedefDeclaration> | ( <parserTypeDeclaration> ; )
    | ( <controlTypeDeclaration> ; ) | ( <packageTypeDeclaration> ; )

<derivedTypeDeclaration> := <headerTypeDeclaration> | <headerUnionDeclaration> |
    <structTypeDeclaration> | <enumDeclaration>

<headerTypeDeclaration> := header <name> { <structFieldList> }

<headerUnionDeclaration> := header_union <name> { <structFieldList> }

<structTypeDeclaration> := struct <name> { <structFieldList> }

<structFieldList> := { <structField> { , <structField> }* }

<structField> := <typeRef> <name> ;

<enumDeclaration> := enum { bit < INTEGER > } <name> { <specifiedIdentifierList> }

<errorDeclaration> := error { <identifierList> }

<matchKindDeclaration> := match_kind { <identifierList> }

<identifierList> := <name> { , <name> }*

<specifiedIdentifierList> := <specifiedIdentifier> { , <specifiedIdentifier> }*

<specifiedIdentifier> := <name> { = <expression> }

<typedefDeclaration> := ( typedef | type ) ( <typeRef> | <derivedTypeDeclaration> ) <name> ';'

```

STATEMENTS

```

<assignmentOrMethodCallStatement> := <lvalue>
    ( { < <typeArgumentList> > } ( <argumentList> ) ; ) |
    ( = <expression> ; )

<emptyStatement> := ;

<returnStatement> := return { expression } ;

<exitStatement> := exit ;

<conditionalStatement> := if ( <expression> ) <statement> { else <statement> }

/* To support direct invocation of a control or parser without instantiation. */
<directApplication> := <typeName> . apply ( <argumentList> ) ;

<statement> := <assignmentOrMethodCallStatement> | <directApplication> | <conditionalStatement> |
    <emptyStatement> | <blockStatement> | <exitStatement> | <returnStatement> | <switchStatement>

```

```

<blockStatement> := { <statementOrDeclList> }

<statementOrDeclList> := { <statementOrDeclaration> }*

<switchStatement> := switch ( <expression> ) { <switchCases> }

<switchCases> := { <switchCase> }*

<switchCase> := <switchLabel> : { <blockStatement> }

<switchLabel> := <name> | default

<statementOrDeclaration> := <variableDeclaration> | <statement> | <instantiation>

```

TABLES

```

<tableDeclaration> := table <name> { <tablePropertyList> }

<tablePropertyList> := <tableProperty> { <tableProperty> }*

<tableProperty> :=
  ( key = { <keyElementList> } ) |
  ( actions = { <actionList> } ) |
  ( const entries = { <entriesList> } ) | /* immutable entries */
  ( { const } <nonTableKwName> = <expression> ; )

<keyElementList> := { <keyElement> }*

<keyElement> := <expression> : <name> ;

<actionList> := { <actionRef> ; }*

<actionRef> := <prefixedNonTypeName> { ( <argumentList> ) }

<entriesList> := <entry> { <entry> }*

<entry> := <keysetExpression> : <actionRef> ;

<actionDeclaration> := action <name> ( <parameterList> ) <blockStatement>

```

VARIABLES

```

<variableDeclaration> := { const } <typeRef> <name> { = <expression> };

```

EXPRESSIONS

```

<functionDeclaration> := <functionPrototype> <blockStatement>

<argumentList> := { <argument> { , <argument> }* }

<argument> := <expression> | _

<expressionList> := { <expression> { , <expression> }* }

<prefixedNonTypeName> := { . } <nonTypeName>

<lvalue> := <prefixedNonTypeName> {
  ( . <name> ) | /* member selector */
  ( [ <indexExpression> ] ) /* array subscript */
}*

<expression> := <expressionPrimary> { <exprOperator> <expression> }*

```

```

<expressionPrimary> := <integer> | <boolean> | <string> |
  ( { . } <nonTypeName> ) |
  ( { <expressionList> } ) |
  ( ( <expression> ) ) |
  ( ( ! | ~ | - ) <expression> ) | /* unary expression */
  ( <namedType> | error ) | /* member selector, function call */
  ( ( <typeRef> ) <expression> ) /* cast */

<exprOperator> := <binaryOperator> |
  ( . <name> ) | /* member selector */
  ( [ <indexExpression> ] ) | /* array subscript */
  ( ( <argumentList> ) ) | /* function call */
  ( < <realTypeArgumentList> > ) |
  ( = <expression> ) /* named argument */

<indexExpression> := <expression> { : <expression> }

<integer> := INTEGER

<boolean> := true | false

<string> := STRING

<binaryOperator> := * | / | + | - | <= | >= | < | > | != | == | || | && | | | & | << | >>

```

Syntax Tree

PROGRAM

```

<p4program> := <declarationList>decl_list

<declarationList> := { <declaration>[0..n] }*

<declaration> := ( <variableDeclaration> | <externDeclaration> | <actionDeclaration> |
  <functionDeclaration> | <parserDeclaration> | <parserTypeDeclaration> | <controlDeclaration> |
  <controlTypeDeclaration> | <typeDeclaration> | <errorDeclaration> | <matchKindDeclaration> |
  <instantiation> )decl

<name> := STRINGstrname

<parameterList> := { <parameter>[0..n] }*

<parameter> := DIRECTIONdirection <typeRef>type <name>name { <expression> }init_expr

<packageTypeDeclaration> := <name>name { <typeParameterList> }type_params <parameterList>params

<instantiation> := <typeRef>type_ref <argumentList>args <name>name

```

PARSER

```

<parserDeclaration> := <typeDeclaration>proto { <parameterList> }ctor_params
  <parserLocalElements>local_elements <parserStates>states

<parserTypeDeclaration> := <name>name { <typeParameterList> }type_params <parameterList>params

<parserLocalElements> := { <parserLocalElement>[0..n] }*

<parserLocalElement> := ( <variableDeclaration> | <instantiation> )element

```

```

<parserStates> := { <parserState>[0..n] }+

<parserState> := <name>name <parserStatements>stmt_list <transitionStatement>transition_stmt

<parserStatements> := { <parserStatement>[0..n] }*

<parserStatement> := ( <assignmentStatement> | <functionCall> | <directApplication> |
    <parserBlockStatement> | <variableDeclaration> )stmt

<parserBlockStatement> := <parserStatements>stmt_list

<transitionStatement> := <stateExpression>stmt

<stateExpression> := ( <name> | <selectExpression> )expr

<selectExpression> := <expressionList>expr_list <selectCaseList>case_list

<selectCaseList> := { <selectCase>[0..n] }*

<selectCase> := <keysetExpression>keyset_expr <name>name

<keysetExpression> := ( <tupleKeysetExpression> | <simpleKeysetExpression> )expr

<tupleKeysetExpression> := <simpleExpressionList>expr_list

<simpleKeysetExpression> := ( <expression> | <default> | <dontcare> )expr

<simpleExpressionList> := { <simpleKeysetExpression>[0..n] }+

```

CONTROL

```

<controlDeclaration> := <typeDeclaration>proto { <parameterList> }ctor_params
    <controlLocalDeclarations>local_decls <blockStatement>apply_stmt

<controlTypeDeclaration> := <name>name { <typeParameterList> }type_params <parameterList>params

<controlLocalDeclarations> := { <controlLocalDeclaration>[0..n] }*

<controlLocalDeclaration> := ( <variableDeclaration> | <actionDeclaration> | <tableDeclaration> |
    <instantiation> )decl

```

EXTERN

```

<externDeclaration> := ( <externTypeDeclaration> | <functionPrototype> )decl

<externTypeDeclaration> := <name>name { <typeParameterList>type_params } <methodPrototypes>method_protos

<methodPrototypes> := { <functionPrototype>[0..n] }*

<functionPrototype> := { <typeRef> }return_type <name>name { <typeParameterList> }type_params
    <parameterList>params

```

TYPES

```

<typeRef> := ( <baseTypeBoolean> | <baseTypeInteger> | <baseTypeBit> | <baseTypeVarbit> |
    <baseTypeString> | <baseTypeVoid> | <baseTypeError> | <name> | <specializedType> |
    <headerStackType> | <tupleType> )type

```

<tupleType> := <typeArgumentList>_{type_args}

<headerStackType> := <name>_{name} <expression>_{stack_expr}

<specializedType> := <name>_{name} <typeArgumentList>_{type_args}

<baseTypeBoolean> := <name>_{name}

<baseTypeInteger> := <name>_{name} { <integerTypeSize> }_{size}

<baseTypeBit> := <name>_{name} { <integerTypeSize> }_{size}

<baseTypeVarbit> := <name>_{name} <integerTypeSize>_{size}

<baseTypeString> := <name>_{name}

<baseTypeVoid> := <name>_{name}

<baseTypeError> := <name>_{name}

<integerTypeSize> := INTEGER_{size}

<typeParameterList> := { <name>[_0..n] }₊

<realTypeArg> := (<typeRef> | <dontcare>)_{arg}

<typeArg> := (<typeRef> | <name> | <dontcare>)_{arg}

<realTypeArgumentList> := { <realTypeArg>[_0..n] }₊

<typeArgumentList> := { <typeArg>[_0..n] }_{*}

<typeDeclaration> := (<derivedTypeDeclaration> | <typedefDeclaration> | <parserTypeDeclaration> |
 <controlTypeDeclaration> | <packageTypeDeclaration>)_{decl}

<derivedTypeDeclaration> := (<headerTypeDeclaration> | <headerUnionDeclaration> |
 <structTypeDeclaration> | <enumDeclaration>)_{decl}

<headerTypeDeclaration> := <name>_{name} <structFieldList>_{fields}

<headerUnionDeclaration> := <name>_{name} <structFieldList>_{fields}

<structTypeDeclaration> := <name>_{name} <structFieldList>_{fields}

<structFieldList> := { <structField>[_0..n] }_{*}

<structField> := <typeRef>_{type} <name>_{name}

<enumDeclaration> := INTEGER_{type_size} <name>_{name} <specifiedIdentifierList>_{fields}

<errorDeclaration> := <identifierList>_{fields}

<matchKindDeclaration> := <identifierList>_{fields}

<identifierList> := { <name>[_0..n] }₊

<specifiedIdentifierList> := { <specifiedIdentifier>[_0..n] }₊

<specifiedIdentifier> := <name>_{name} { <expression> }_{init_expr}

<typedefDeclaration> := (<typeRef> | <derivedTypeDeclaration>)_{type_ref} <name>_{name}

STATEMENTS

<assignmentStatement> := (<expression> | <lvalueExpression>)_{lhs_expr} <expression>_{rhs_expr}

<functionCall> := (<expression> | <lvalueExpression>)_{lhs_expr} <argumentList>_{args}

<returnStatement> := { <expression> }_{expr}

<exitStatement> := **exit**

<conditionalStatement> := <expression>_{cond_expr} <statement>_{stmt} { <statement>_{else_stmt} }

<directApplication> := (<name> | <typeRef>)_{name} <argumentList>_{args}

<statement> := (<assignmentStatement> | <functionCall> | <directApplication> |
 <conditionalStatement> | <emptyStatement> | <blockStatement> | <exitStatement> |
 <returnStatement> | <switchStatement>)_{stmt}

<blockStatement> := <statementOrDeclList>_{stmt_list}

<statementOrDeclList> := { <statementOrDeclaration>_[0..n] }*

<switchStatement> := <expression>_{expr} <switchCases>_{switch_cases}

<switchCases> := { <switchCase>_[0..n] }*

<switchCase> := <switchLabel>_{label} { <blockStatement>_{stmt} }

<switchLabel> := (<name> | <default>)_{label}

<statementOrDeclaration> := (<variableDeclaration> | <statement> | <instantiation>)_{stmt}

TABLES

<tableDeclaration> := <name>_{name} <tablePropertyList>_{prop_list}

<tablePropertyList> := { tableProperty_[0..n] }+

<tableProperty> := (<keyProperty> | <actionsProperty> | <entriesProperty> | <simpleProperty>)_{prop}

<keyProperty> := <keyElementList>_{keyelem_list}

<keyElementList> := { <keyElement>_[0..n] }*

<keyElement> := <expression>_{expr} <name>_{match}

<actionsProperty> := <actionList>_{action_list}

<actionList> := { <actionRef>_[0..n] }*

<actionRef> := <name>_{name} { <argumentList>_{args} }

<entriesProperty> := <entriesList>_{entries_list}

<entriesList> := { <entry>_[0..n] }+

<entry> := <keysetExpression>_{keyset} <actionRef>_{action}

<simpleProperty> := <name>_{name} <expression>_{init_expr}

<actionDeclaration> := <name>_{name} <parameterList>_{params} <blockStatement>_{stmt}

VARIABLES

<variableDeclaration> := <typeRef>_{type} <name>_{name} { <expression> }_{init_expr}

EXPRESSIONS

<functionDeclaration> := <functionPrototype>_{proto} <blockStatement>_{stmt}

<argumentList> := { <argument>[_0..n] }*

<argument> := (<expression> |)_{arg}

<expressionList> := { <expression>[_0..n] }*

<lvalueExpression> := (<name> | <memberSelector> | <arraySubscript>)_{expr}

<expression> := (<expression> | <booleanLiteral> | <integerLiteral> | <stringLiteral> | <name> | <specializedType> | <headerStackType> | <expressionList> | castExpression | <unaryExpression> | <binaryExpression> | <memberSelector> | <arraySubscript> | <functionCall>)_{expr}
{ <realTypeArgumentList> }_{type_args}

<castExpression> := <typeRef>_{type} <expression>_{expr}

<unaryExpression> := OPERATOR_{op} <expression>_{operand}

<binaryExpression> := <expression>_{left_operand} OPERATOR_{op} <expression>_{right_operand}

<memberSelector> := (<expression> | <lvalueExpression>)_{lhs_expr} <name>_{name}

<arraySubscript> := (<expression> | <lvalueExpression>)_{lhs_expr} <indexExpression>_{index_expr}

<indexExpression> := <expression>_{start_index} { <expression> }_{end_index}

<booleanLiteral> := INTEGER_{value}

<integerLiteral> := INTEGER_{value} INTEGER_{width}

<stringLiteral> := STRING_{value}

<default> := **default**

<dontcare> := **_**